CACHE-CRAFT: Managing Chunk-Caches for Efficient Retrieval-Augmented Generation

Shubham Agarwal[1*], Sai Sundaresan[1*], Subrata Mitra[1†], Debabrata Mahapatra[1]
Archit Gupta[2‡], Rounak Sharma[3‡], Nirmal Joshua Kapu[3‡], Tong Yu[1], Shiv Saini[1]
[1]Adobe Research [2]IIT Bombay [3]IIT Kanpur

Abstract

Retrieval-Augmented Generation (RAG) is often used with Large Language Models (LLMs) to infuse domain knowledge or user-specific information. In RAG, given a user query, a retriever extracts chunks of relevant text from a knowledge base. These chunks are sent to an LLM as part of the input prompt. Typically, any given chunk is repeatedly retrieved across user questions. However, currently, for every question, attention-layers in LLMs fully compute the key values (KVs) repeatedly for the input chunks, as state-of-the-art methods cannot reuse KV-caches when chunks appear at arbitrary locations with arbitrary contexts. Naive reuse leads to output quality degradation. This leads to potentially redundant computations on expensive GPUs and increases latency. In this work, we propose CACHE-CRAFT, a system for managing and reusing precomputed KVs corresponding to the text chunks (we call *chunk-caches*) in RAG-based systems. We present how to identify *chunk-caches* that are reusable, how to efficiently perform a small fraction of recomputation to *fix* the cache to maintain output quality, and how to efficiently store and evict *chunk-caches* in the hardware for maximizing reuse while masking any overheads. With real production workloads as well as synthetic datasets, we show that CACHE-CRAFT reduces redundant computation by **51%** over SOTA prefix-caching and **75%** over full recomputation. Additionally, with continuous batching on a real production workload, we get a **1.6**× speedup in throughput and a **2**× reduction in end-to-end response latency over prefix-caching while maintaining quality, for both the LLaMA-3-8B and LLaMA-3-70B models.

1 Introduction

Retrieval-Augmented Generation (RAG) allows LLMs to access relevant context from a custom knowledge base outside its training data to generate grounded responses. RAG systems do not require model retraining and are therefore a cost-effective way to customize an LLM's output such that it is relevant and accurate with respect to a target knowledge base. The key components of a RAG-based system are a vector database and an LLM. The vector database stores the embedding of text chunks from a specific domain as indexes. During the *retrieval* phase, relevant chunks are extracted based on these embeddings, using vector-similarity search [25, 26].

In the *generation* phase, the LLM uses the retrieved context to generate a response to the user's question. The LLM processes its input prompt (retrieved chunks + user's question) in the `prefill` phase, building an initial computed-state called *Key-Value* cache (KV-cache), which is then used in the `decode` phase for autoregressive token generation. The prefill phase is compute-bound because it processes all tokens of the input prompt in parallel; while the decode phase, which generates one token at a time, is memory-bound.

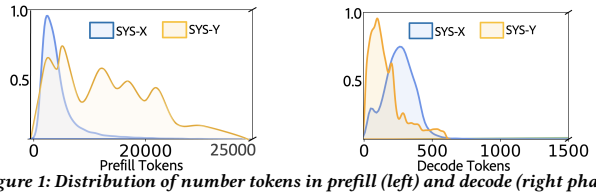


Figure 1: Distribution of number tokens in prefill (left) and decode (right phases for two real production RAG systems SYS-X ***and*** SYS-Y.

We aim to understand the bottlenecks in these systems. In this regard, we use two real production RAG systems, referred to as SYS-X and SYS-Y, for workload characterization, motivation, and evaluations. SYS-X helps users in setting up complex workflows for an enterprise SaaS product by answering queries and providing steps from user manuals and SYS-Y helps users search and understand concepts from large knowledge bases through repeated Q&A.

Computational bottlenecks in RAG-systems: Typically for RAG systems, the answers generated as well as user questions are short. However, longer input context with more relevant information is often crucial for the system to generate a well-informed answer [16]. We highlight this in Fig. 1 we show distributions of prefill (left) and decode (right) tokens for SYS-X and SYS-Y. It can be seen, that the number of prefill tokens is much more.

The prefill time increases quadratically with the length of input context, due to the attention computation in the transformer architecture [73]. Fig. 2 shows prefill time increases with input token length across different batch sizes for LLaMA-3-70B [71] using vLLM [42] with 4 NVIDIA A100-80GB GPUs, reaching up to **76 seconds** for 32k-token sequences at a batch size 8. In real production workloads, the prefill time can often cross more than 100 seconds when serving multiple concurrent users. This increases the time-to-first-token (TTFT) [5] and degrades the user experience, as no response is generated until the whole input context is processed. The impact of the prefill phase on overall latency is significant. In SYS-X, it accounts for up to 77% of total inference time.

This problem is further exacerbated by the emergence of new LLMs, that can consume up to 1 million tokens (e.g., Claude 3 [9] and Gemini [64]). As more chunks can be used to improve response quality, longer context would lead to even longer TTFT.

Opportunities for optimizing prefill in RAG: RAG systems typically operate on a finite knowledge base [45]. Moreover, our analysis, shown in Fig. 3 for SYS-X and RAG datasets like 2WikiMQA and MuSiQue, reveals that a subset of chunks gets retrieved frequently by the system. For SYS-X, 75% of the retrieved chunks for a query were reprocessed, amounting to over 12B tokens in a month. Processing these tokens would require 9600 hours of GPU compute on LLaMA-3-70B using 8 A100 GPUs, costing approximately $50k.

Challenges in KV-cache reuse in RAG: Indiscriminate reuse of KV-caches from previously processed parts of the knowledge base can disrupt the relative positions of tokens, violating causal attention and degrading output quality [73]. Additionally, for RAG

*Equal contributions. † Corresponding Author (subrata.mitra@adobe.com).
‡Work done at Adobe Research.

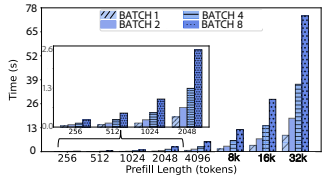


Figure 2: Prefill time across prefill length and batch size in vLLM on A100 80GB with TP=4.

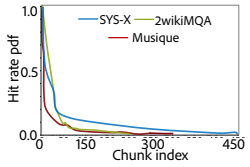


Figure 3: Chunk-cache hit rate pdf for both Sys-X and RAG datasets.

systems with large knowledge bases, precomputed KV-caches may not remain in GPU memory, as space is required for storing a) the LLM parameters and b) the growing KV-cache during the decode phase. The latency of loading precomputed KV-caches into GPU memory must not negate the savings from bypassing recomputation, requiring efficient system design and implementation.

Limitations of existing works: Recent efforts to reduce prefill time and cost, such as *Paged Attention* [42], *CacheGen* [53], *Radix Attention* [84], *RAG cache* [40] and *context caching* in *Gemini* [64] rely on *prefix caching*, where different prompts with an identical prefix share KV-cache. While this preserves the output quality by maintaining *causal attention* [73], its usefulness is very limited in RAG systems because the RAG-system-retrieved text chunks and their relative ordering are sensitive to the input question. Slight variation in the user question can result in different sets of chunks and ordering, rendering the prefix caching technique ineffective. We found in production workloads, exact prefix caching applies to only a small fraction (8%) of requests and 18% of total prefill tokens.

Cache-Craft: We propose Cache-Craft, a system for managing and reusing precomputed KV-caches in RAG. We overcome the challenges by (1) efficiently identifying which *chunk-caches* can be reused even if their prefix alters, (2) identifying how to recompute the KV of a few selected tokens of the prefix-altered-caches to prevent quality degradation, and (3) how to manage these caches such that most important chunks are prioritized, the overhead of load/store is masked to effectively reduce expensive GPU compute and TTFT latency for a workload. Fig. 4 illustrates Cache-Craft:

- On the left, we show how KV-caches are formed across the `Transformer` layers when attention computation is done on the text chunks (shown with *yellow* and *gray*) corresponding to a question Q. These pre-computed *chunk-caches* are stored and managed by Cache-Craft along with some metadata.
- On the right we show their reuse. For a *new question* two chunks of the knowledge-base become important. Cache-Craft identifies that it already has a cache for the *yellow* chunk, therefore it retrieves and reuses the caches at the appropriate layers and only computation for the new *green* chunk happens across layers.
- When a *chunk-cache* is reused, KV is recomputed (not shown here) for a limited number of *tokens* that were originally contextualized by tokens outside the chunk. Cache-Craft further reduces this recomputation by using the relevance of a chunk w.r.t. the new question. Tokens of less relevant chunks are not recomputed beyond a certain number of layers.
- Cache-Craft prioritizes storing KV-caches for the *important* chunks to maximize computation savings. The importance of a chunk is determined by its potential for direct reuse without significant recomputation, as well as its expected frequency of use based on the RAG's workload.

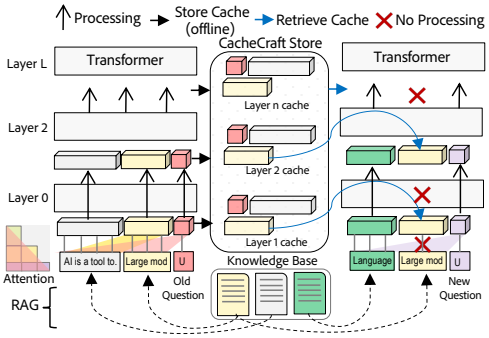


Figure 4: Overview of Cache-Craft

We implement Cache-Craft and integrate its KV-cache management capabilities into vLLM [42], a widely used package for LLM inference. The implementation is non-trivial as it incorporates optimizations such as *FlashAttention* [17] and *PagedAttention* [42] to enhance the *Arithmetic Intensity* [60] of computations.

We evaluate Cache-Craft with LLaMA in real deployment scenarios based on public traces. We show that it achieves a 51% reduction in GPU computation costs for production workloads compared to prefix-caching (§5.4). Under continuous batching through *ORCA* for Sys-X, Cache-Craft improves throughput by 1.6× and reduces end-to-end response latency by 2.1× for LLaMA-3-8B model and for LLaMA-3-70B, it provides a 1.6× speedup in throughput and 2× reduction in end-to-end response latency compared to prefix-caching (§5.3). In both cases, 30% tokens are recomputed which maintains 90% of the base ROUGE F1 score on average.

In summary, this paper makes the following contributions:

(1) We analyze real production workloads to show that RAG systems are prefill-heavy, yet prefix caching remains ineffective.
(2) We present the key challenge of reuse, stemming from causal attention calculation through a formal problem formulation, and present detailed techniques to identify the reusability of *chunk-caches* along with an efficient recomputation strategy to fix any potential degradation in generation quality.
(3) We present end-to-end design details and rationale for Cache-Craft, which is our optimized KV-cache management system for RAG, implemented in vLLM, a widely used LLM inference package and plan to open-source it.
(4) We present extensive evaluations on real-world, large production RAG systems, along with six other datasets, supported by a human evaluation user study and several sensitivity studies.

2 Background and Motivation

2.1 Preliminaries of LLM

A transformer-based LLM progressively contextualizes a sequence of tokens $S = \{t_1, \cdots, t_n\}$ using L *transformer* layers. Each layer $l \in [L]$ receives $d-$dimensional embeddings of n tokens, $H^l \in \mathbb{R}^{n \times d}$, as input, and outputs contextualized embeddings $H^{l+1} \in \mathbb{R}^{n \times d}$, which are known as hidden states. We denote the LLM operations, from input tokens S all the way up to the last hidden states H^L, as $H^L(S)$. The last layer hidden states are used in a task-specific manner. In the text generation task, the hidden embedding of the last token $H^L_n(S) \in \mathbb{R}^d$ is used to predict the $(n+1)^{\text{th}}$ token.

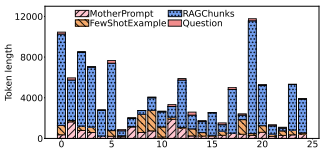
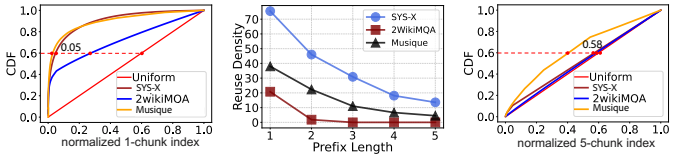


Figure 5: Token distribution of different prompt components (Mother prompt, RAG chunks, Examples, Query, etc.) across RAG use cases.



(a) Individual chunks *(b) Reuse Density* *(c) Observed 5-tuples*

Figure 6: Fig. 6(a) and 6(c) show the CDF of retrieval hit rates of the individual chunks and the observed 5−tuple chunks respectively, across all user requests. Fig. 6(b) shows the decreasing cache reuse density with increasing prefix lengths.

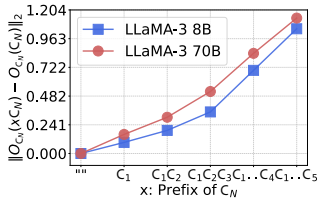


Figure 7: Deviation in output for chunk C_1 with increasing prefix chunks

In l^{th} transformer layer, first, the H^l is linearly transformed into the Query, Key, and Value matrices, $Q, K, V \in \mathbb{R}^{n\times d}$ respectively. The Q and K matrices are further transformed by positional embeddings (either absolute [69] or relative [48]) to capture the sequential order of the tokens. Then the attention mechanism contextualizes the value embedding of j^{th} token as $\tilde{V}_j = \text{softmax}(Q_j K_{:j}^T)V_{:j}$, where $Q_j \in \mathbb{R}^{1\times d}$ is the j^{th} query vector and $K_{:j}, V_{:j} \in \mathbb{R}^{j\times d}$ are all the key and value vectors up to the j^{th} token. Finally, the contextualized hidden state H_j^{l+1} is obtained by normalizing $H_j^l + \text{FNN}(\tilde{V}_j)$.

LLM operates in two phases. In the **prefill** phase, it contextualizes all available prompt tokens. The hidden states $H^L(S)$ are computed for the prompt $S = \{t_1, \cdots, t_n\}$ using the matrix operation

$$\tilde{V} = \text{softmax}(QK^T \odot M)V, \tag{1}$$

where $\odot$ denotes element-wise product, and $M \in \{0,1\}^{n\times n}$ is a lower triangular matrix, known as causal attention mask, to ensure each token attends only to its previous tokens. This attention computation is $O(n^2)$, as both Q and K matrices are of size $n \times d$. During this phase, the model generates the KV pairs for all tokens in the sequence, which are used to predict the next token t_{n+1}.

In the **decode** phase, the model generates tokens autoregressively. For each newly generated token t_j starting from the position $j = n + 1$, the attention mechanism is applied to contextualize its raw embedding H_j^0. However, instead of recomputing the KV for all previous tokens again, the model uses the cached K and V matrices (KV cache) from the prefill phase. By reusing these cached representations of the previous n tokens at every layer, the computation is reduced from $O(n^2)$ to $O(n)$. As each new token is generated, the KV cache is updated by adding the new token's key and value.

KV cache vs. Prefix cache: While KV cache optimizes the decode phase by reusing KV pairs of previously processed tokens, the prefill phase still requires $O(n^2)$ computation to establish the full context. Prefix-cache stores and reuses previously computed context that matches with the prefix of the input prompt, and only computes the rest [84] to reduce prefill computation.

2.2 Prefill Dominates Decode in RAG

In a typical RAG-system, the overall prompt sequence S consists of a few initial instruction text chunks, several retrieved chunks from a knowledge base, and the user's question or request U, i.e., $S = C_1 : C_k U$, where $C_1 : C_k$ denotes the concatenation of k chunks.* In most production RAG systems, between 5 to 15 chunks are retrieved to answer a query U. The overall length of prefill tokens $|S|$ and the lengths of their constituents may vary for different requests. We analyze this in Fig. 5 for a proprietary system, Sys-X, from 25 sessions. The majority of the tokens (60% to 98%) are from the retrieved chunks from a knowledge base (in blue). A few tokens are from the mother prompt (instructions for the chatbot), few-shot examples (for in-context learning [21]), and the user's questions.

Due to the extra chunks apart from the user's question, the number of prefill tokens becomes significantly more than that of decode. To verify this, we analyze three systems: proprietary production Sys-X and Sys-Y, and another open-source LMSys [83] chat system. Fig. 1 shows a disparity in the number of tokens between prefill and decode: **30k** prefill tokens for **600** decode tokens on average.

We compare prefill times and operations on 4xA100-80GB GPUs using the LLaMA-70B. For Sys-X, prefill accounts for 55.4% of total time and 19.3x decode operations. Sys-Y takes 76% of the time and 46x the operations, while LMSys uses 22% time and 4.4x operations.

Contrary to the popular belief that decode is slow in LLMs, for RAG-systems prefill phase typically dominates both the amount of token computation and total latency, despite being highly parallelized.

2.3 Evidences of Chunk-Reuse

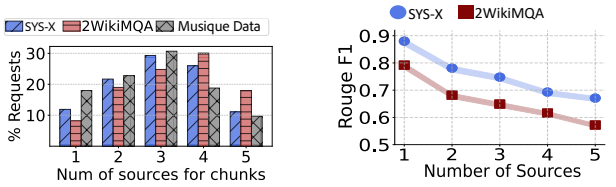
Since prefill is the primary bottleneck in RAG, we find improvement opportunities by observing repetitions in chunk retrieval. If N is the total number of chunks representing the knowledge base accessible for RAG, a significant portion of N is retrieved multiple times across different user sessions, where a session consists of multiple user requests and LLM responses.

We substantiate this by analyzing the *retrieval hit rates*, defined as the fraction of all retrievals (across multiple sessions) in which a particular chunk is present. Fig 5a shows the retrieval hit rates of 3 RAG systems: Sys-X, 2wikiMQA [35] and MuSiQue [72]. The top 5% of chunks are accessed by **60%** of the requests in both Sys-X and MuSiQue, and **40%** requests in 2wikiMQA. In Sys-X, most chunk reuse occurs across users (94%), with reuse within a session at 55% and across sessions at 67%. Exploiting the high reuse of knowledge chunks can optimize the prefill by reusing caches that are computed in the previous sessions, instead of recomputing for every retrieval. However, cache reuse across different user requests is non-trivial.

2.4 Why Cache-Reuse is non-trivial?

Limitations of Prefix-Caching: The prefix-cache approach is to store the KV-caches of ordered k-tuple chunks when they are co-retrieved for a request U, and reuse it for a future request U' if the same k chunks are retrieved in the same order. However, the *reuse density*, defined as the number of ordered $k-$tuple chunks observed in previous requests, drops significantly w.r.t. k, reducing the reusability of their cache. We analyze reuse density for 3 datasets

*The instructions in the prompt are the same across all prompts. These instructions are similar to an always repeated chunk and can be dealt with under the same framework.



(a) (b)

Figure 8: (a) Shows the distribution of chunk sources for non-prefix chunk reuse, while (b) Shows the impact of chunk reuse from multiple sources on ROUGE F1 scores, even when using new positional embeddings.

over the most recent 1000 requests in Fig. 6b, and find that it drops to as low as 5 for $k = 5$. This worsens further as we analyze the 5−tuple retrieval hit rates, defined as the fraction of all observed 5−tuple retrievals in which a particular 5− tuple is present. Fig. 6c shows this hit rate is significantly low compared to those of the individual chunks in Fig. 6a. However, a high hit rate is crucial for cache utilization. Moreover, unlike Fig. 6a, the distribution in Fig. 6c does not follow the power law, indicating that, for a high re-usability, several k-tuple chunks should be cached. Therefore, the combinatorial growth of the number of possible k-tuple makes the memory footprint of prefix-caching prohibitive.

On the other hand, a method that can reuse the KV-cache of individual chunks (pre-computed while serving a past request) at any position, without restricting to the prefix order, would have a significantly high retrieval hit rate as evidenced in Fig. 6a.

Although a high hit rate is encouraging, it presents a few major decision challenges. We lay them out in the following.

Contextualization: A chunk C may have more than one stored KV-caches that were computed while serving different user requests in the past, e.g., $S^1 = C_1^1 : C_i : C_k^1 U^1$ and $S^2 = C_1^2 : C_j : C_k^2 U^2$, where $C = C_i = C_j$, but the positions i and j are not necessarily the same. Which of these KV caches of C should be used for a new request U^3? A key challenge with reusing pre-computed KV-cache is contextualization. The stored KV-cache of C_i have been contextualized by $C_1^1 : C_{i-1}^1$. We analyze how the contextualization of C_i changes with varying numbers of prefix chunks. In particular, we use the last layer hidden states $H_{C_i}^L(C_1 : C_i) \in \mathbb{R}^{|C_i| \times d}$ corresponding to the tokens in C_i, when $C_1 : C_i$ is given as input. Fig. 7 shows its difference from that of no contextualization $H^L(C_i)$. Evidently, the contextualization grows with more prefix chunks.

Sensitivity to chunk ordering: The relative ordering of prefix chunks affects the contextualization due to two reasons: a) the unidirectional attention by the causal attention mask M in (1) and b) the positional embedding that alters Q and K matrices specific to the token positions. More precisely, $H_{C_i}^L(C_1 : C_i) \neq H_{C_i}^L(C_{(1)} : C_{(i-1)} C_i)$, where $C_{(1)} : C_{(i-1)}$ is a permutation of the prefix chunks.

Cache from multiple sources: Another decision challenge occurs when the KV-caches are stored at different requests. Let C and C' are retrieved for serving U, the KV-cache of C was stored from a past prompt $S_1 = C_1^1 : C : C_k^1 U^1$ and that of C' was stored from a different prompt $S_2 = C_1^2 : C' : C_k^2 U^2$. In such cases, can any of the chunk's KV-cache be used reliably to serve the new request U? Can both be used? Fig. 8a shows that for a majority of the requests in Sys-X and 2wikiMQA, the 5 retrieved chunks were found in the retrievals of 3 past requests. Finding all the 5 chunks in the retrievals of only 1 past request is not common (around 10% for Sys-X and 8% for 2wikiMQA). A naive reuse of KV-caches that were precomputed across different requests significantly degrades output quality. Our findings in Fig. 8b show a **50%** drop in F1 score when all 5 chunks are reused from five distinct past requests (Fig. 8b), highlighting the need for a more advanced reuse strategy.

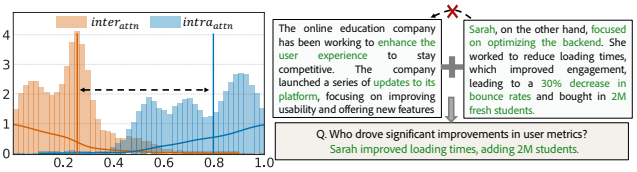


Figure 9: Inter-attention (C1, C2) and intra-attention (C2, C2) distributions for chunks C2 with C1 in context. The overlap in the distribution is less, meaning inter<intra, and hence the output for <C1, C2, Q> without letting C2 attend to C1 is correct due to less overlap indicating little contextualization.

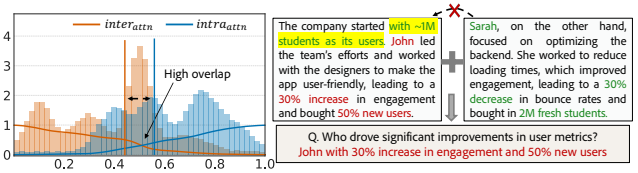


Figure 10: Inter-attention (C1, C2) and intra-attention (C2, C2) distributions for chunks C2 with C1 in context. The overlap in the distribution is more, meaning inter$\not<$intra, and hence the output for <C1, C2, Q> without letting C2 attend to C1 is incorrect due to more overlap indicating contextualization.

To understand when naive reuse of the KV-cache works and when it does not, we analyze two example prompts, and their outputs in Figs. 9 and 10. We use $k = 2$ relevant chunks C_1, C_2 to construct the prompt of a question U. The KV-caches of C_1 and C_2 are precomputed from $H^L(C_0 C_1)$ and $H^L(C_0' C_2)$ respectively. We observe in Fig. 10 that when the values of intra-chunk attention weights (from $Q_{C_2} K_{C_2}^T$) and inter-chunk attention weights (from $Q_{C_2} K_{C_1}^T$) are highly overlapping, naive reuse of stale KV-cache results in a wrong output. Whereas if they are less overlapping, the precomputed KV-cache can lead to the right answer in Fig. 9.

3 Cache-Craft Design

At a high level, Cache-Craft enhances a RAG system by managing the KV-caches of knowledge chunks, as illustrated in Fig.4. We denote the *Chunk-Cache* of a chunk C that was originally computed from $H^L(C_1 : C_i : C_k U)$, while serving a request U at the i^{th} position (i.e., $C_i = C$), as

$$\mathscr{C}(C \mid C_1 : C_{i-1}) := \left\{ \left(K_C^l, V_C^l \right) \,\middle|\, l \in [L] \right\}, \tag{2}$$

where K_C^l and V_C^l are the key and value vectors in l^{th} layer corresponding to the tokens in C.

Apart from storing the $\mathscr{C}(C \mid C_1 : C_{i-1})$, We also store certain metadata to determine whether a particular of KV-cache C can serve a new request U' in future. Cache-Craft operates in two phases: online and offline. The metadata computation is performed in the offline phase, and the determination of its "usefulness" is performed in the online phase, while serving a new request U'.

In its online phase, Cache-Craft first selects the most useful (w.r.t. U') version of *chunk-cache* of C out of all the stored versions $\mathscr{C}(C| \cdots)$. Then Cache-Craft selectively recomputes the key and value vectors for a few tokens of C to contextualize w.r.t. U'. Clearly, if there are no *chunk-caches* of C, then the key and value vectors for

all tokens have to be computed afresh. Once the K and V matrices of C are contextualized for U', either by fixing a stored chunk or by computing afresh, CACHE-CRAFT repeats this same online procedure for the chunk next to C that is inline to serve U'.

3.1 Determining Cache Reusability

From our analysis in Fig. 9 and 10, we observe that reusability can be assessed by determining how much a chunk's KV computation is influenced by external context (tokens outside the chunk) versus its own tokens. If a chunk is mainly influenced by its own tokens, it is more likely to produce high-quality answers when reused.

In fact, a chunk with more tokens is more reusable because tokens closer to each other have stronger attention due to positional embeddings, compared to distant tokens from other chunks [69]. To capture the attention within and across the chunks, we define the following two attention-based metrics:

(1) **Inter attention** measures the cumulative attention weight from tokens in chunk C_i to tokens in chunk C_j where $i < j$:

$$inter(C_i, C_j) = \sum_{k \in C_i} \sum_{l \in C_j} a_{kl}, \tag{3}$$

where a_{kl} is the *attention weight* from the k^{th} token of chunk i to the l^{th} token of chunk j, computed from the softmax in (1).

(2) **Intra attention** measures the cumulative attention weight within chunk C_i from each token to previous tokens in the same chunk:

$$intra(C_i) = \sum_{k,l \in C_i : k<l} a_{kl}. \tag{4}$$

The attention weights involved in the *inter* and *intra* are used to obtain the output from the attention computation. For instance, in case of 3 chunks $[C1, C2, C3]$, the attention output is

$$\begin{bmatrix} \tilde{V}_{C_1} \\ \tilde{V}_{C_2} \\ \tilde{V}_{C_3} \end{bmatrix} = \begin{bmatrix} \overline{intra}(C_1) & \mathbf{0} & \mathbf{0} \\ \overline{inter}(C_1, C_2) & \overline{intra}(C_2) & \mathbf{0} \\ \overline{inter}(C_1, C_3) & \overline{inter}(C_2, C_3) & \overline{intra}(C_3) \end{bmatrix} \begin{bmatrix} V_{C_1} \\ V_{C_2} \\ V_{C_3} \end{bmatrix}, \tag{5}$$

where $\overline{intra}$ and $\overline{inter}$ represent the associated attention weights in (3) and (4) without summing them, V_C represents the pre-attention value vectors of all tokens in chunk C, and $\tilde{V}_C$ represents the corresponding post-attention value vectors.

Reusability of the cache $\mathscr{C}(C_3|C_1C_2)$ for a new request depends on the new prefixes. Consider 2 cases with prefixes (i) C_4-C_2-C_3 and (ii) C_5-C_6-C_3. The first sequence carries the C_2 as a prefix similar to that of $\mathscr{C}(C_3|C_1C_2)$, making it more reusable than the second sequence, which does not have any common chunk in its prefix.

Assuming that the higher the prefix overlap, the higher will be the reusability, we calculate a *Prefix Overlap Score* β for a *chunk-cache* of C_i corresponding to the current prompt sequence S_{new} as:

$$\beta(C_i \mid S_{new}) = \frac{\sum_{j \in S_{old} \cap S_{new}} inter(i, j)}{\sum_{j \in S_{old}} inter(i, j)}, \tag{6}$$

where S_{old} is the set of chunks forming C_i's old prefix.

However, since β simply sums the *inter* attention terms for overlapping chunks, it is order-invariant and only captures the subset match between the previous and current prefixes. For instance, consider the two scenarios: C_1-C_2-C_3 and C_2-C_1-C_3. In both cases, β equals 1, yet the potential for reusing the cached chunk C_3 can differ significantly due to the reordering of the prefix sequence.

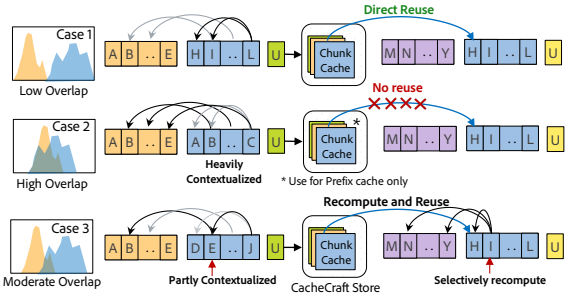


Figure 11: Chunk-cache reuse scenarios. Inter and intra-attention for the blue chunk are shown on the left. Dark arrows represent high contextualization, and gray arrows indicate low. Case 1: Blue chunk is self-contextualized, so cache can be used even with new context with purple chunk. Case 2: Blue chunk is heavily contextualized on outside orange chunk, no reuse. Case 3: Only few tokens of blue are contextualized outside, so can be reused with selective recomputation.

Note, it is not prudent to manipulate the retrieved order of chunks to match the prefix of the cached-chunk, due to *lost-in-the middle* phenomena with LLM-based RAG systems [51].

To account for prefix reordering, we introduce the *Order Penalty Score* (γ), which penalizes a chunk for different ordering in the prefix sequence. Let $A_{old} = \langle C_i \mid C_i \in S_{old} \cap S_{new} \rangle$ denote the ordered sequence of chunks according to S_{old}'s order, and similarly A_{new}. We define γ for chunk C_i w.r.t. S_{new} as the normalized Kendall's Tau distance [13] between vector A_{old} and A_{new}:

$$\gamma(C_i \mid S_{new}) = \frac{D}{T}, \quad T = \binom{m}{2} = \frac{m(m-1)}{2}, \tag{7}$$

where $m = |S_{old} \cap S_{new}|$ and D is the number of discordant pairs between A_1 and A_2. A higher value of D indicates a greater discrepancy in ordering, leading to a higher penalty for reuse. Hence, we adjust β to account for this discrepancy by penalizing it, resulting in the *Adjusted Prefix Overlap Score* denoted by

$$\beta'(C_i \mid S_{new}) = \beta(C_i \mid S_{new}) \cdot (1 - \gamma(C_i \mid S_{new})). \tag{8}$$

Finally, to assess the reusability of a chunk across different prefix contexts, we measure how much the chunk's KV is contextualized by its prefix. A chunk's KV is more reusable if it is a) less influenced by its prefix and b) more influenced by its own tokens. We formulate these two effects by calculating as:

$$a(C_i) = \sum_{j<i} \frac{inter(C_j, C_i)}{|C_i| \cdot |C_j|} \quad \text{and} \quad b(C_i) = \frac{intra(C_i)}{|C_i|^2}, \tag{9}$$

where a is the normalized sum of *inter*-attention scores between chunk C_i and its prefix chunks at the time of caching, and b is the normalized *intra*-attention score of chunk C_i. Normalizing w.r.t. the chunk length $|C|$ ensures comparability across chunks of varying sizes. The layer-wise inter and intra values are averaged to

$$\bar{a}(C_i) = \frac{1}{L} \sum_{l=1}^{L} a_l(C_i) \quad \text{and} \quad \bar{b}(C_i) = \frac{1}{L} \sum_{l=1}^{L} b_l(C_i). \tag{10}$$

A higher $\frac{\bar{a}}{\bar{b}}$ ratio indicates greater outside contextual influence on the chunk's KV. We use this ratio to define the *Cache Context Impact (CCI)* for chunk C_i as

$$CCI(C_i) = \frac{1}{1 + e^{-\frac{\bar{a}}{\bar{b}}}}, \tag{11}$$

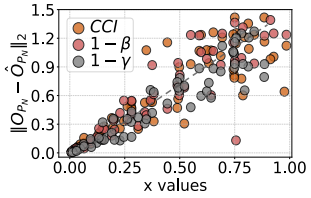


Figure 12: Output deviation with increasing CCI, 1-β and 1-γ

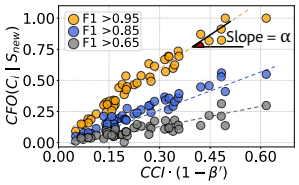


Figure 13: CFO as a function of CCI and $1-\beta'$ to find α in Eq. 12

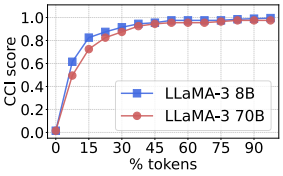


Figure 14: CCI score is majorly from top recomp candidates

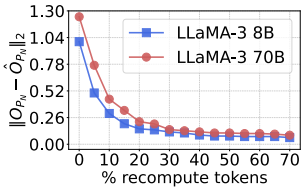


Figure 15: Output deviation decreases with higher recomputation

where the `sigmod` function standardizes its range between 0 and 1. A high value of the *CCI* for a cache indicates that the chunk is highly contextualized, reducing its potential for reuse unless the prefix context matches closely. Conversely, a low *CCI* suggests that the chunk is largely independent of its prefix context, making it more reusable across different contexts. Fig. 11 shows three different scenarios to illustrate how *chunk-cache* is reused by CacheCraft. Inter- and intra-attention for the blue chunk are shown on the left and dark/black arrows for high contextualization, while gray arrows are for low contextualization. **Case 1:** The blue chunk is self-contextualized, allowing cache reuse even with the new purple chunk context. **Case 2:** The blue chunk is highly contextualized by the orange chunk, so no reuse is possible. **Case 3:** Only a few tokens in the blue chunk are contextualized externally, allowing partial reuse with selective recomputation.

3.2 Fixing Chunk-Cache via Recomputation

We *fix* the *chunk-caches* at runtime to make them reusable across different contexts. Fixing refers to recomputing only the necessary KV values to ensure the output of the reused cache closely mimics the output without any cache reuse. Fig. 12 shows how output deviations increase with higher *CCI* and higher $1-\beta'$, indicating greater fixing requirements. *CCI* captures the chunk's contextual dependency, while $1-\beta'$ reflects prefix mismatch. We use this to define Cache Fix Overhead (*CFO*) for chunk C_i as

$$CFO(C_i \mid S_{new}) = \alpha \cdot CCI \cdot (1-\beta'), \tag{12}$$

where α is a scaling hyperparameter that adjusts the recomputation factor. A higher value of *CFO* indicates a higher fraction of the tokens in C_i needs KV recomputation: $CFO = 1$ for recomputing all tokens in C_i.

Setting α in deployment: As we lower the value of α, and corresponding CFO_α, in expectation we would employ less recomputation per request. This might lead to corresponding quality score ($F1_\alpha$) to go down below the acceptable level ($F1_{desired}$). We determine α from a validation dataset by solving:

$$\alpha^* = \arg\min_{\alpha} \mathbb{E}[\text{CFO}_\alpha], \quad \text{subject to} \quad \text{F1}_\alpha \geq \text{F1}_{\text{desired}}. \tag{13}$$

Fig. 13 plots *CFO* against $CCI \cdot (1-\beta')$ for different α values and F1 scores on 2WikiMQA [35] dataset.

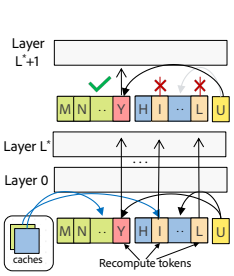


Figure 16: Algorithm selects "focused" Chunks across layers to reduce recomputation

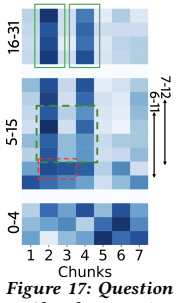


Figure 17: Question to Chunks attention across layers

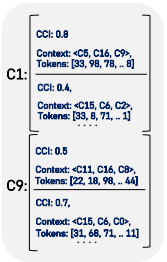


Figure 18: Metadata Store for "chunk-cache" lookup

3.2.1 Token Selection for Recomputation: We have observed that a small subset of tokens in a chunk significantly impacts the *CCI* score (Fig. 14). Also, recomputing these critical tokens reduces output deviation (Fig. 15). Hence, to reuse a *chunk-cache*, we focus on recomputing the top $N = \lceil CFO(C_i) \cdot |C_i| \rceil$ tokens with the highest inter-attention scores from prior chunks. We select the top-N contextualized tokens for chunk C_i as

$$\mathbb{T}(C_i) = \arg top_N \Big(\Big\{ \sum_{j<i} inter(C_j, t_k) \Big\}_{t_k \in C_i} \Big), \tag{14}$$

where $inter(t_k, C_j)$ denotes the inter-attention score between token t_k in chunk C_i and a prefix chunk C_j. This method ensures the selection of the most contextualized tokens for recomputation.

3.2.2 Adaptive Early Recomputation Termination. In RAG pipelines, it is a standard practice to retrieve a sufficient number of chunks from a large knowledge base and utilize the LLM to filter out irrelevant content for coherent answers [76]. In CacheCraft, we leverage this characteristic to reduce runtime recomputation costs.

In each layer l, during the recomputation of selected tokens, we monitor the attention between a chunk C_i and the current question U, i.e., $inter_l(C_i, U)$, to identify the chunks that consistently receive *"focused"* attention from U as shown in Fig. 16. We find that while the inter-attention scores vary during the initial layers, after a certain number of layers, they settle into values that can segregate the focused chunks from the others. Fig. 17 illustrates that for approximately 80% of queries, focused chunks can be detected between layers 10 and 15 for the `LLaMA-3-8B` model. Consequently, we early-terminate the recomputation of tokens in the *"unfocused"* chunks to minimize unnecessary computations.

Algorithm 1 details our method for predicting focused chunks, drawing ideas from change-point detection [8]. In each a layer l, we first calculate the inter-attention scores w.r.t. the user question U for each chunk cumulated up to layer l:

$$cinter_l(C_i, U) = \sum_{l'=1}^{l} inter_{l'}(C_i, U) \quad \text{for all } i \in [k]. \tag{15}$$

Then based on these cumulative scores, we segregate the high-valued chunks from the low-valued ones in an adaptive manner (lines 5-9). If this set of high-valued chunks does not change for w consecutive layers, then we deem them as the focused chunks and stop the recomputation for other chunks.

In §6 (Fig. 26), we observe that this approach reduces token recomputation by about 55% while maintaining similar output quality.

Algorithm 1 Predicting Focused Chunks

Require: L: total number of layers, w: layer confidence window
Ensure: F^*: set of focused chunks, L^*: recomputation cut-off layer

$F_{all} \leftarrow [],\ cinter_i \leftarrow 0 \quad \forall i \in [k]$
for $l = 1$ to L **do**
  $cinter_i \mathrel{+}= inter_l(C_i, U) \quad \forall i \in [k]$
  $sorted_cinter \leftarrow$ **sort**$([cinter_i \mid i \in [k]]$, descending$)$
  $diff \leftarrow [sorted_cinter_i - sorted_cinter_{i+1} \mid i \in [k-1]]$
  $p_i \leftarrow \frac{diff_i}{\sum_{i=1}^{k-1} diff_i} \quad \forall i \in [k-1]$
  $h_i \leftarrow -\sum_{j=0}^{i} p_j \cdot \log(p_j) \quad \forall i \in [k-1]$
  $i^* \leftarrow$ **argmax**$([h_{i+1} - h_i \mid i \in [k-2]])$
  F = top i^* chunks in $sorted_cinter$
  F_{all}.append(F)
  if $l \geq w$ & is_all_equal($F_{all}[l-w:l]$) == 1 **then**
    $F^*, L^* \leftarrow F, l$
    Break
return $F^*,\ L^*$

3.3 Cache Variants: Retrieval and Eviction

Cache-Craft maintains a data structure, as shown in Fig. 18, for efficient lookup, retrieval, and eviction. Each *chunk-cache* is identified by hashing the original chunk texts linked to the RAG vector similarity search (§1). This results in a `map` where chunk hashes serve as keys and lists of prefixes for each chunk are stored as values. Cache-Craft targets to store $N \times M$ *chunk-cache* instances, starting with N chunks (the number of keys in the `map`), each having M variants. These variants help Cache-Craft recover from cases where the initial *chunk-cache* may not be optimal (e.g., excessive token recomputation due to high contextualization), while subsequent *chunk-cache* variants may be more reusable for common contexts. Each variant stores the *CCI* value and an ordered list of token indices needing recomputation. To find the best *chunk-cache* for a request, Cache-Craft calculates the reusability score $CFO = CCI \times (1 - \beta')$ (as discussed in § 2.3) and selects the variant with the lowest score to minimize token recomputation.

For each *chunk-cache* access, Cache-Craft updates its *frequency-reuse* (f_r) as $f_r \mathrel{+}= 1/CFO$. Consequently, *chunk-caches* with higher prefix matches or less contextualization become more reusable, as indicated by increasing f_r over time. New variants are added when Cache-Craft encounters a unique chunk and prefix until it reaches $N \times M$ instances. After this, Cache-Craft periodically evicts caches with the lowest f_r to make room for more effective variants. This allows diverse configurations, from one popular chunk with $N \times M$ variants to $N \times M$ chunks, each with a single variant.

This design enables Cache-Craft to manage storage dynamically, prioritizing caches that maximize reusability while minimizing recomputation, thus reducing prefill computation. Traditional policies like LRU, LFU, or FIFO do not offer this capability. The choice of M and N is influenced by the popularity and reusability of the *chunk-caches*, the RAG setting (i.e., the number of retrieved chunks), the architecture (GPU/CPU memory size and interconnects), and the deployment configuration of the LLM.

3.4 Chunk-Cache Reuse Pipeline

Cache-Craft implements an efficient LLM inference pipeline to minimize redundant computations in RAG by strategically reusing *chunk-caches* across prefill requests.

3.4.1 Recomputation Planning: For a user query U, the prefill request consists of ordered chunks $C_1, C_2, \ldots, C_n$ provided by RAG. The system first queries the *Metadata Store*, a CPU-memory-based hash-table, to determine which chunks have their *chunk-caches* available. Based on this, the chunks are then classified into two subsets: C_{hit} (with *chunk-caches*) and C_{miss} (without *chunk-caches*).

It then generates an Inference Plan, designating chunks in C_{miss} for *chunk-cache* computation and those in C_{hit} for *chunk-cache* retrieval. It uses the metadata retrieved from the *Metadata Store* to compute the *Adjusted Prefix Overlap* score (β') and the *Chunk Context Impact* score (*CCI*) and then determines the *Cache Fixing Overhead* (*CFO*) (§3.2). Finally, the *top-N contextualized tokens* $\mathbb{T}$, that need to be recomputed for each C_{hit} *chunk-cache* are identified.

Note that both the *Metadata Store* and vLLM's KV-block manager are distinct CPU-memory hash-tables. While the *Metadata Store* tracks RAG chunk metadata, the KV-block hash-table maps tokens to their respective KV-blocks. Details on enabling independent chunk access without relying on prior prefixes are provided in §4.

3.4.2 Layer-wise Preloading of cache-chunks: Cache-Craft uses a hierarchical caching mechanism across GPU memory, CPU memory, and SSD to expand the effective memory capacity available to store the *chunk-caches*. To reduce average loading latency, the most frequently used *chunk-caches* are stored in GPU, less frequently ones in CPU, and infrequent ones in SSD. Further, Cache-Craft uses a layer-wise preloading technique to minimize cache loading delays. While the GPU processes layer l, the *chunk-caches* for the layer $l+1$ are concurrently loaded from host memory or SSD. Specifically, Cache-Craft overlaps the loading of caches for C_{hit} chunks for layer $l+1$ with two activities: a) prefill computation of new C_{miss} chunks and b) KV recomputation of tokens in C_{hit} chunks in layer l. This ensures that by the time the GPU begins computing attention for layer $l+1$, the corresponding *chunk-caches* are already available in the execution buffer.

However, preloading may not fully overlap with computation if the *chunk-cache* loading time exceeds the computation time for a layer, particularly when loading from SSDs. To address this, Cache-Craft reserves an HBM read buffer that allows preloading *chunk-caches* for multiple layers in advance. We determine the optimal preloading depth L_p as:

$$L_p = (L-1)\left(1 - \frac{T_{prefill}}{T_{load}}\right) + 1, \tag{16}$$

where L is the total number of layers, $T_{prefill}$ is prefill computation time and T_{load} is KV loading time. The goal is to preload L_p layers such that the *chunk-caches* for the remaining $(L - L_p)$ layers can be loaded within the computation time for $(L-1)$ layers.

Algorithm 2 shows how layer-wise preloading is implemented. When $T_{load} > T_{prefill}$, preloading L_p layers minimizes wait times by eliminating layer execution gaps. If $T_{prefill} \geq T_{load}$, preloading just one layer is sufficient due to the longer prefill time. Fig. 19 shows an example for $L = 5$ layers with a $T_{prefill} : T_{load}$ ratio of 1:2 where preloading $L_p = 3$ layers eliminates execution gaps.

3.4.3 Handling Partial Prefill in LLM: For each layer, the Key-Value pairs (KV) for *chunk-caches* are fetched from HBM, while the K, V, and Q are computed only for new chunks and recomputation tokens. To support *chunk-cache* reuse across contexts, Cache-Craft

Algorithm 2 Layer-wise Preloading of Chunk-Caches

Require: L: Total layers, T_{prefill}: Prefill time, T_{load}: Load time
Ensure: Minimized execution gaps
1: $L_p \leftarrow \max(1, (L-1) \cdot (1 - T_{\text{prefill}}/T_{\text{load}}) + 1)$
2: **for** $i = 1$ to L **do**
3: **for** $j = i$ to $\min(i + L_p, L)$ **do**
4: Preload layer j
5: Compute layer i
6: Release resources for layer i

decouples Rotary Position Embedding (RPE) from K in KV caches. This allows dynamic RPE application during inference, adapting *chunk-caches* to new positions. The system first merges the retrieved and newly computed KV, then applies the new RPE to the merged K based on updated chunk positions, and also applies RPE to the computed Query Q.

Next, attention is computed using the newly computed Q and the merged K and V. Since Q has a different shape than K and V, a custom attention mask is required, replacing the standard triangular causal mask. As shown in Fig. 4, *attention scores* (QXK^T) are computed with this custom mask, multiplied by V, and passed through the feedforward network (FFN) to produce the layer output.

During attention computation, new *chunk-caches* and inter/intra attention (QXK^T) for the new chunks are asynchronously saved in the background. Additionally, at every layer, attention output between question and RAG chunks (Q^l_{inter}) is used for *Focused Chunk Selection* (§3.2.2). Once the focused chunks are determined at layer L^*, recomputation for "unfocused" chunks stops. Note that the Q for the last prefill token is always computed to generate the first decode token. After prefill, the decode phase proceeds as usual with all KV data—cached, newly computed, and recomputed.

3.5 Hierarchical Chunk-Cache Management

CACHE-CRAFT manages cache storage efficiently across GPU High Bandwidth Memory (HBM), host (CPU) memory, and SSD so that less frequent caches are moved to further locations (from GPU HBM-memory to SSD) without deleting them when the LLM requires more GPU memory.

To offset the loading time of caches from non-HBM locations, CACHE-CRAFT employs preloading techniques that start to move the caches to GPU memory, asynchronously, while requests are still in the queue. If the caches are available in GPU memory when the request is ready to be executed, CACHE-CRAFT uses it; otherwise, it defaults to prefill from scratch starting from input text tokens. This technique ensures that highly reusable chunks remain in HBM while low-reuse chunks are progressively swapped to CPU-memory, and later to SSD, before eventual eviction, if not reused.

Using such asynchronous as well as layer-wise (§ 3.4.2) preloading, CACHE-CRAFT significantly reduces loading delay to make chunk-caching effective. For example, the loading time required for 5 RAG *chunk-caches* corresponding to a request in SYS-X takes **0.03s** for CPU and **0.59s** for SSD. In SYS-X a typical queue wait time is **0.32s**, allowing for preloading chunks from CPU or SSD without impacting latency significantly. For higher loads, queue time can completely mask the loading time even from the SSD.

Cache Scaling and Workload Adaptability: In production workloads, the *chunk-cache* size grows with question diversity

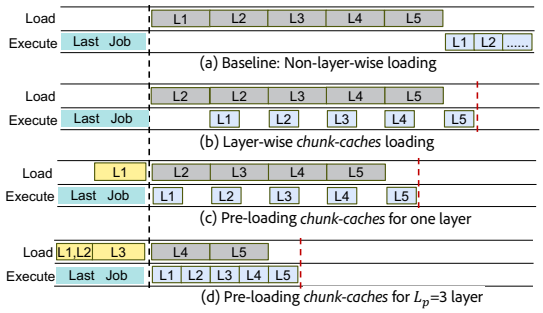


Figure 19: Layer-wise preloading of chunk-caches into GPU memory to eliminate wait time during prefill execution.

but a small set of chunks remains crucial. For example, in SYS-X, 20.6% of chunks were accessed over 1 month, with 85% of requests hitting only 13.5% of chunks. By 3 months, chunk accesses increased to 24%, but 85% of requests still accessed just 14.8%, indicating minimal growth in the required chunk set. In this workload, these stable highly reused chunks (119 GB for LLaMA-3-70B) fit comfortably within the 135 GB free GPU memory budget (corresponding to LLaMa-3-70B hosted on 4, A100-80GB GPUs [12], with Tensor-Parallelism). Hence, our design ensures we can handle growth in *chunk-cache* size in the future.

4 Implementation

CACHE-CRAFT is a wrapper around vLLM [42], built on Xformers [44] backend optimized with Triton [70]. It enables *chunk-cache* reuse for prefix and non-prefix tokens by efficiently managing positional shifts and enabling partial recomputation of prefill.

Chunk Storage Management: CACHE-CRAFT manages *chunk-caches* by implementing a hash table at the granularity of individual RAG chunks. Unlike vLLM, which hashes entire prefixes pointing to the start of the KV cache (spanning multiple chunks), our approach generates independent hashes for each chunk, allowing direct access without dependence on prior context. Each chunk maps to a list of 16-token memory blocks for efficient and independent access. For optimized retrieval, the hash table stores address pointers across memory tiers, prioritizing faster tiers while allowing fallback to slower ones when necessary. Variable chunk sizes are padded to align with 16-token blocks, ensuring a consistent memory layout. Such padding causes negligible output deviation.

RPE Management: To enable the reuse of *chunk-caches* in arbitrary positions, CACHE-CRAFT stores all cached chunks without RPE and dynamically applies corrected positional embeddings during runtime based on the current context. To efficiently manage large caches, CACHE-CRAFT employs a custom CUDA kernel to remove RPE from the Keys of the KV cache after processing each request. This kernel reverses the RPE operation, $x\cos(\theta) - y\sin(\theta)$, by applying its inverse, $y\cos(\theta) + x\sin(\theta)$, where x and y represent the upper and lower 64-dimensional components of each token's 128-dimensional embedding and θ is the rotational angle. CACHE-CRAFT applies relative positional encoding (RPE) to cached chunks before attention computation and removes it after decoding, ensuring reusability across varying positions. In batched inference, it optimizes RPE handling by considering shared chunk positions within the batch. For requests with differing chunk positions, RPE is integrated directly into the attention mechanism during the prefill

and decoding stages. For identical positions, RPE is applied before attention and removed post-decoding. This minimizes computational overhead while ensuring correct positional embedding.

Selective Token Recomputation: Cache-Craft modifies the Triton-based flash attention kernel to enable selective token recomputation during the prefill phase. It computes QKV values for both the scattered recomputed tokens and new question tokens. The attention kernel processes queries from these tokens, performing attention computations with the entire KV matrix and parallelizing operations over recompute and question tokens. During block-wise attention, query blocks are multiplied with prior KV blocks, adhering to causal constraints enforced by a dynamic attention mask, ensuring recompute tokens attend only to preceding tokens. After the prefill phase, we inject corrected KV values for recomputed tokens into vLLM's cache for autoregressive decoding. To prevent cache corruption, the updated cache is asynchronously swapped with the original after decoding.

5 Evaluation

5.1 Experimental Set up

5.1.1 System Configuration: We evaluate Cache-Craft on the LLaMA-3 8B and 70B models [24] with tensor parallelism (TP) of 1 and 4 respectively. All our experiments are performed on EC2 p4de.24xlarge [1] instances with 8 A100 GPUs [12] with each having 80 GB GPU (HBM) memory. The host CPU is an Intel Xeon Platinum 8275L processor with 48 cores (96 vCPUs). The instance has 1152 GB of main memory and an 8 TB NVMe SSD with a read throughput of 16 GB/s. The CPU and GPUs are interconnected via PCIe 4.0 ×16, providing 64 GB/s bandwidth.

5.1.2 Datasets and Workload: We evaluate our technique with a real production RAG workload (Sys-X) as well as relevant datasets following previous works[10, 40].

(1) **Real-world workloads:** Sys-X helps users set up complex workflows for an enterprise SaaS by answering questions and prescribing steps from user manuals. It retrieves top-k=5 chunks based on the query. As Sys-X creates a chunk based on the subsections of the user manual, each of the chunks can have a highly variable number of tokens. This results in a total input size of 1k-20k tokens with a median of 3.3k tokens (Fig. 5a).
(2) **Single-Hop QnA:** A question can be answered from a single chunk for this class of datasets. SQuAD [63] focuses on extracting answers from passages, while DROP [23] requires discrete reasoning over chunks. For multi-chunk RAG with $k = 5$, we selected 200 questions and split them into 512-token chunks.
(3) **Multi-Hop QnA:** This class of datasets requires using facts and data from multiple chunks to answer each question properly. We utilize 2WikiMQA [35] and MuSiQue [72], which are benchmarks for evaluating complex answers across multiple documents. We sampled 200 questions.
(4) **Summarization:** We use CNN dataset [58] that generates summaries of news articles from CNN, and XSUM [59] that focuses on single-sentence summaries from BBC. For sampling, we split long chunks into smaller segments and randomly selected top-k=5 chunks. This method is applied to 40 large chunks, resulting in 200 summarization tasks.

Cache Warm-Up: For every dataset, we use the first 20 queries to warm up the system and set up the caches so that we can evaluate the steady-state characteristics.

Cache Storage: We store $N = 100$ chunks with $M = 5$ variants, requiring 0.05 TB for LLaMA-3-8B model. Specifically, caching 100 chunks, each containing around 1000 tokens across 5 versions, consumes $100 \times 1000 \times 5 \times 0.1$ MB (per token) = 50 GB. For LLaMA-3-70B, this increases to 150 GB with 0.3 MB cache per token.

Tasks: In the Single-Hop and Multi-Hop QnA datasets, we perform both long and short answering tasks by adjusting the mother prompt, i.e., instructing the LLM. Additionally, we generate 200 True/False questions from the original dataset chunks. For the Summarization task, we focus solely on long summaries.

5.1.3 Evaluation Metrics: We use two quality metrics: ROUGE-L F1 [49], which measures long-answer quality in Single/Multi-Hop and summarization tasks, and Jaccard Similarity [37], which is used for short answers and True/False questions. We also conduct a *user-study* with 250 participants to assess response correctness and quality on 2wikiMQA and SQuAD datasets based on Yes/No ratings.

Note, according to several prior studies [11, 50], a ROUGE-L F1 score ≥ 0.6 is considered good, and a score ≥ 0.8 is considered almost indistinguishable from the original answer. From our user study, we also analyzed this correlation and found that for answers with ROUGE-L F1 scores ≥ 0.6 and ≥ 0.8, 81% and 93% of users have given a YES, respectively. For efficiency, we measure Recompute Savings, Time-to-First-Token (TTFT i.e., prefill latency), System Throughput, and Cost Savings.

5.1.4 Baselines We evaluate against the following baselines.

(1) **Prefix Matching**: We compare with two methods: (1) **Prefix-Cache** [42], which reuses the KV cache based on exact prefix matches. While this approach offers perfect accuracy, it has low reuse potential. (2) **Set-Cache**, which modifies RPE to reorder *chunk-caches* and finds the longest exact prefix match with the query. While this provides higher reuse, it has *lower accuracy*.
(2) **Naive KV Reuse (Full-Cache)**: This baseline reuses the KV cache for each chunk irrespective of the previous context, fixing only the RPE at the new position. No recomputation is performed for the chunks.
(3) **Recomputation Strategies**: We also evaluate against recomputation methods: (1) **Random-Recomp**, which randomly recomputes tokens within each chunk, and (2) **Prefill-H2O** [82], which recomputes the most-attended tokens in the chunks. For both strategies, we maintain the same average fraction of recomputed tokens as in Cache-Craft.
(4) **Full Recomp (Full-Recomp)**: This **oracle** baseline fully recomputes all chunks for a request without utilizing any cache, providing a benchmark for optimal performance.
(5) **Compression Techniques**: We compare with prefill compression methods: (1) **Lingua2** [38] that reduces prefill length by discarding less significant tokens using a trained model (e.g., GPT-2), and (2) **MapReduce** [18] that summarizes context chunks for compression. The compression rates are aligned with Cache-Craft's recomputation, where 80% compression corresponds to 20% recompute.

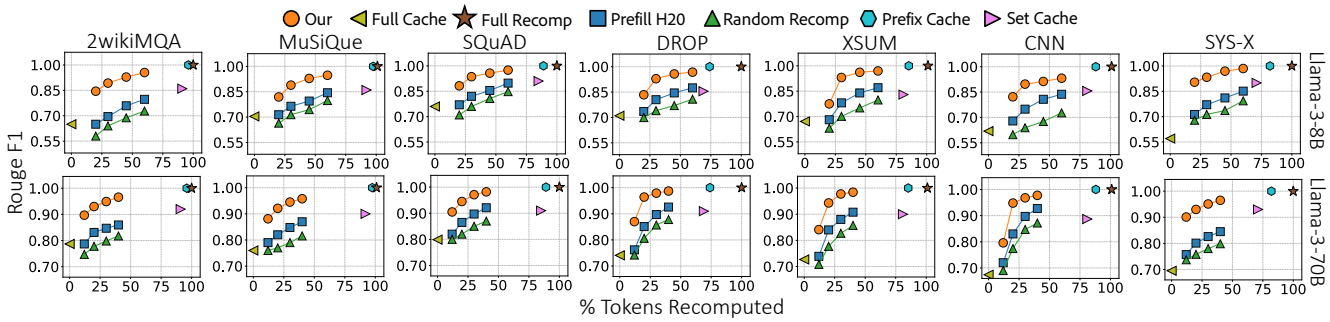


Figure 20: Rouge F1 of answer generated using Llama-3 8B and 70B on multi-hop QA, single-hop QA, text summarization, and on production Sys-X.

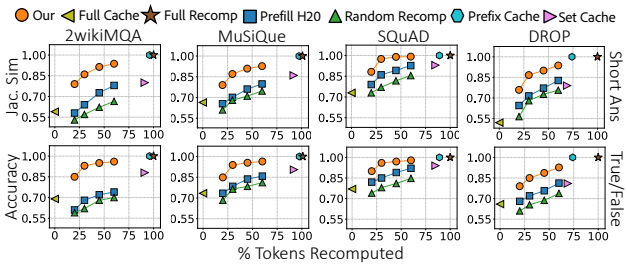


Figure 21: Jaccard Similarity and Accuracy of short answers and True/False generated using Llama-3 8B on multi-hop and single-hop datasets.

5.2 Generation Quality with KV Chunk Reuse

5.2.1 Evaluation of Recomputation Strategy We evaluate the recomputation strategy of Cache-Craft for Question Answering (Long and Short), True/False, and Summarization tasks using LLaMA-3-8B and LLaMA-3-70B models across multiple datasets.

Fig. 20 shows ROUGE-F1 scores, comparing Cache-Craft with baseline KV-cache reuse techniques and the original LLaMA generation (i.e., Full-Recomp with ROUGE score=1). Using Full-Cache incurs no recomputation but yields low quality, ROUGE dropping to 0.65 for multi-hop QA datasets like 2wikiMQA and MuSiQue. In contrast, recomputing 20% of tokens with LLaMA-3-8B improves the ROUGE by 30%, and further by 42% with 30% recomputation. This trend is consistent across single-hop QA and summarization datasets, with ≈20-35% improvements for both 8B and 70B models. Moreover, increasing recomputation to 45% and 60% for LLaMA-3-8B, and 30% and 40% for LLaMA-3-70B, further improves ROUGE scores, reaching within 1–5% of Full-Recomp across all datasets.

We also compare our contextualization-based recomputation against Random-Recomp (random token selection) and Prefill-H2O (high-attention token selection). Notably, random selection can lower performance even below Full-Cache as it neglects the key contextual tokens and overpowers wrong tokens, which can even shadow/underpower crucial ones. Prefill-H2O shows only a modest 2-10% improvement over Full-Cache but struggles with multi-hop tasks. Cache-Craft identifies and recomputes critical tokens distorted by prior contexts, enhancing performance and minimizing missing or incorrect facts. Fig. 21 further shows that Cache-Craft outperforms Full-Cache by up to 50% in short-QA and True/False tasks, achieving ROUGE of 0.87, compared to 0.59.

Prefix-Cache offers exact answers, but due to low prefix match rates, 80-95% of tokens go through regular KV-computation, leading to very low compute savings. Set-Cache gives slightly more savings of 15-35% by making prefixes permutation invariant with lower ROUGE due to incorrect contextualization.

To summarize, Cache-Craft offers the ***best trade-off*** as its points on Fig. 20 are the furthest towards the ***top-left (ideal)*** corner.

Table 1: ROUGE-F1 scores comparing Cache-Craft with token compression techniques for 30% recompute tokens using LLaMA-3-8B

Dataset	Lingua2	MapReduce	Our
2wikiMQA	32.1%	53.0%	89.3%
SQuAD	45.2%	63.6%	93.6%
XSUM	51.2%	51.6%	91.1%
Sys-X	56.4%	61.0%	92.0%

Table 2: User study comparing Cache-Craft (30% recomp) with Full Cache, Prefill H20 and Full Recompute on Llama-3-8B.

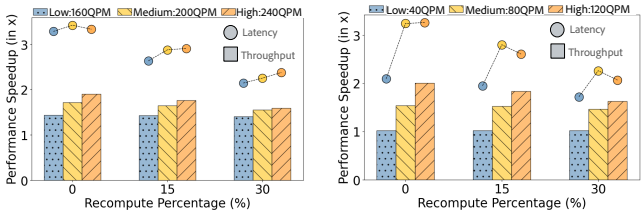
	2wikiMQA	SQuAD
Full-Cache	29.8%	53.1%
Prefill-H2O	52.4%	66.8%
Our	71.2%	78.9%
Full-Recomp	76.9%	83.7%

5.2.2 Comparison with Prompt Compression Techniques We compare Cache-Craft with established context reduction methods such as Lingua2 [38] and MapReduce [18], using datasets for multi-hop (2wikiMQA), single-hop (SQuAD), and summarization (XSUM), along with real-workload from Sys-X, on for LLaMA-3-8B. In Table 1, it can be observed that with 30% recomputation, Cache-Craft gives ROUGE-F1 scores around 0.9, which is ≈100% higher than the scores for Lingua2 (0.4) and MapReduce (0.5), for 70% compression (i.e. comparable to 30% recomputation). The performance gap is due to Lingua2 and MapReduce's approach of discarding tokens, often losing critical information. In contrast, Cache-Craft retains all tokens by leveraging *chunk-cache* reuse, ensuring no context is lost. Additionally, Cache-Craft selectively recomputes the most contextually impacted tokens balancing efficiency and quality.

5.2.3 Cache-Craft on Real Production RAG Workload: We evaluate Cache-Craft on production RAG workloads from Sys-X focused on retrieval-based QA tasks where questions span multiple subsections of user manuals. As shown in Fig. 20, Cache-Craft achieves a ROUGE score of 0.87 with only 20% token recomputation, outperforming Full-Cache reuse (0.59) and other recomputation strategies by about 20-30%. We also see that Prefix-Cache also saves just 18% prefill tokens, proving ineffective. Table 1 further compares Cache-Craft with prompt compression techniques, where Lingua2 (0.56) and MapReduce (0.61) score significantly lower on the Sys-X dataset.

5.2.4 User Study: We conducted a user study with 250 participants with two datasets: 2wikiMQA and SQuAD.

Task: We sampled 500 questions from each dataset, extracted 5 relevant chunks as the context from the datasets using vector-similarity search, and then generated answers using four methods: (1) pass



(a) Cache-Craft performance on LLaMA-3-8B on 1 A100-80GB for Sys-X *(b) Cache-Craft performance on LLaMA-3-70B on 4 A100-80GB for* Sys-X

Figure 22: Throughput and overall system response latency speedup under varying computational loads for Cache-Craft deployed with vLLM using ORCA.

the text tokens of the chunks and the question to LLaMA-3-8B for a Full-Recomp to get the answers, (2) use Full-Cache that simply reuses the *chunk-caches* with LLaMA-3-8B without any recompute (3) Prefill-H20 allows *chunk-caches* reuse by recomputing the heavily attended 30% tokens and (4) Cache-Craft that decides which *chunk-caches* to reuse and which tokens to recompute with 30% recomputation. Finally, each participant was presented with 15 questions, the relevant context, and answers generated by one of the methods and asked to mark Yes/No, based on the correctness and quality of the answers, when compared to the context.

As illustrated in Table 2, Full-Cache achieved significantly lower scores (30% on 2wikiMQA, 53% on SQuAD), whereas Cache-Craft consistently outperformed it (71% on 2wikiMQA, 79% on SQuAD). We also tested an alternative recomputation strategy, Prefill-H20, which showed moderate improvement (52% on 2wikiMQA, 67% on SQuAD), outperforming Full-Cache but still lagging behind Cache-Craft. It is interesting to observe that even answers from LLaMA-3-8B without any *chunk-cache* reuse (Full-Recomp) did not get 100% Yes, it got 77% on 2wikiMQA, 83% on SQuAD which is only marginally better than Cache-Craft. The preference for Cache-Craft was also statistically significant (p-*value* < 0.05).

5.3 Performance Evaluation in Deployment

We evaluate throughput and overall response latency under *continuous batching* through *ORCA* [79]. In continuous batching, instead of waiting for all requests in a batch to complete before starting a new batch, it continuously schedules a new request when a request in the processing batch completes and slots are available. We use Sys-X workload for both LLaMA-3 8B and 70B models on A100-80GB GPUs with a TP of 1 and 4, respectively. The maximum number of batched tokens in *ORCA* is set to 150k tokens. The workload arrival patterns are based on public traces from [7, 66] (which is based on Twitter traces) and proprietary data traces from Sys-X.

5.3.1 Throughput and Response Latency with Continuous Batching: As shown in Fig. 22, Cache-Craft achieves up to a 1.9× speedup in throughput and a 3.3× reduction in response latency under a heavy load of 240 QPM (Queries per minute) for the LLaMA-3-8B model and for LLaMA-3-70B, it provides a 2× speedup in throughput and a 3.3× reduction in response latency under a similar heavy load of 120 QPM with no recomputation. With 30% token recomputation, maintaining 90% of the base ROUGE F1 score on average, we still observe a 1.6× speedup in throughput and a 2.1× reduction in response latency for LLaMA-3-8B and for LLaMA-3-70B, the improvement is a 1.6× speedup in throughput and a 2× reduction in response latency under high load. Notably, a 30% recomputation level for LLaMA-3-70B is sufficient to ensure a minimum of 90% of the base ROUGE F1 score. The overall response latency reduction for LLaMA-3-70B under high load with 30% recomputation is 2.07×, compared to 2.26× under medium load. This difference arises due to the significantly higher wait time overhead at high load (≈7.15s on average) compared to medium load (≈2.15s on average). However, when excluding request wait time, the latency reduction at high load is even more pronounced (3.22× compared to 2.64×).

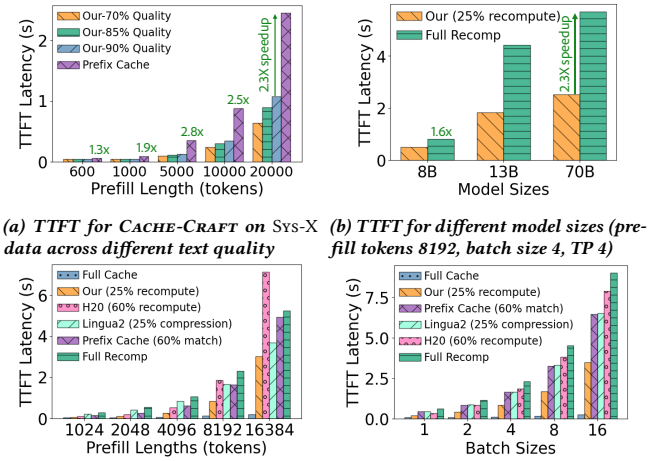
5.3.2 Preloding in Hierarchical Caching: We evaluate how asynchronous (§3.5) and layer-wise (§3.4.2) preloading help in hierarchical caching. In Fig. 29 we show the timings to load the cache from CPU and SSD to GPU-memory when we start loading after the request reaches the head of the queue (Sync), when asynchronous preloading starts when the request is in the queue (Async), and when layer-wise preloading is used with Async (Layer). Cache loading takes 0.03s from the CPU and 0.59s from the SSD, adding significant overhead. With an average queue wait time of 0.32s (for Sys-X), asynchronous preloading eliminates CPU overhead and reduces SSD overhead to 0.27s, as 0.32s overlaps with queue time. Through layer-wise preloading, loading only the first 24 (out of a total of 32) layers in advance further reduces SSD overhead to 0.12s. CPU loading overhead is 0s for Async and Layer because loading time is already less than queue time. We compare our effective prefill time with the time taken to recompute the entire context in the fastest scenario, i.e. when the system is idle. With layer-wise preloading, our effective prefill time is shorter for both CPU and SSD. Note that for caches stored in GPU memory, there is no additional overhead for loading. In all three cases, the cache is brought to the GPU, and the time required for any retrieval from GPU memory for processing is already included in the TTFT.

5.4 Controlled Evaluations for TTFT

We evaluate the TTFT Latency of the vLLM implementation of Cache-Craft on the Sys-X production workload. We also evaluate across various generation settings to ensure generalization to different models and datasets. We compare performance for the setting for which Cache-Craft and the baselines achieve the same quality of ROUGE F1 score of 0.85. For Cache-Craft, Lingua2, and Prefill-H2O, this corresponds to 25% recomputation, 25% compression, and 60% recomputation, respectively. For Prefix-Cache, we copious scope by setting that 60% of the prefill tokens will have a prefix match. Note that this is significantly higher than what (18%) we observed for production workloads Sys-X.

5.4.1 Performance of Cache-Craft on Sys-X*:* In Fig. 23a, we compare the performance of Cache-Craft against Prefix Cache for requests from Sys-X across sequence lengths. The range of sequences received by the system varies from 600 to 20000 tokens averaging around 5000 tokens. We observe about 2.5× speedup for LLaMA-3 8B in TTFT latency over Prefix Cache by recomputing 39% of tokens while maintaining 90% of the original quality. This is because on average only 18% exact prefix match occurs for the requests received in the system rendering Prefix-Cache ineffective.

In Fig. 24, we show TTFT latencies of each request from a trace of the requests received overtime by Sys-X. We indicate the warm-up



(a) TTFT for Cache-Craft on Sys-X data across different text quality

(b) TTFT for different model sizes (prefill tokens 8192, batch size 4, TP 4)

(c) Baseline comparison across prefill lengths (batch size 4, TP 1)

(d) Baseline comparison across batch sizes (prefill tokens 8192, TP 1)

Figure 23: Performance evaluation of Cache-Craft on LLaMA-3-8B (except sizes)

period on the left. In the bottom plot, we observe that as Cache-Craft can keep the TTFT spikes significantly lower than vanilla LLaMA with Prefix-Cache. Cache-Craft provides 3× reduction in the 99th percentile TTFT latency. Note, the spikes in TTFT are due to the fact that text chunks in Sys-X are subsections of the user manuals and are unequal in size. However, note that when prefill lengths are high (leading to spikes), Cache-Craft comes can reduce TTFT significantly as by reusing *chunk-caches*, it avoids quadratic computational complexity (§2). In the top plot of Fig. 24, we also observe a consistent reduction in token computation indicating higher reusability of cached chunks compared to Prefix-Cache. This trend is further supported by the increasing chunk hit rate achieved by our system. This results in a 51% average reduction in token computation compared to Prefix-Cache. In the middle plot of Fig. 24, we show how many chunks for the top-k=5 retrieval in Sys-X was a hit in the *chunk-cache*. It can be observed for a large number of requests, all the necessary chunks were already in the cache — leading to a *hit-rate of 5 out of 5*.

Cache-store Characteristics: Fig. 25 illustrates the cache-store state at the trace's end for Sys-X. The X-axis represents the number of unique *chunk-caches* (186), and the Y-axis indicates how many variants were created for each chunk (up to 11 for some). As detailed in § 3.3, Cache-Craft dynamically configures cache storage based on chunk popularity and reuse.

5.4.2 Impact of Model Size, Prefill Length & Batch Size: Here we measure how Cache-Craft reduces TTFT latency compared to Full-Recomp across different model sizes of LLaMA, across different prefill-lengths, and batch sizes. As shown in Fig. 23b, Cache-Craft becomes more effective in reducing TTFT latency as the model size increases. This improvement is due to reduction of the number of tokens computed by Cache-Craft in each attention layer, with the gains increasing as the number of layers grows for larger models. Cache-Craft reduces latency by 1.6× and 2.3× compared to Full-Recomp for LLaMA-3-8B and LLaMA-3-70B, respectively and with batch-size= 4 and sequence length of 8192 tokens.

In Fig. 23c we compare the TTFT latency against baselines, Prefix-Cache, Lingua2, and Prefill-H2O, across varying sequence lengths

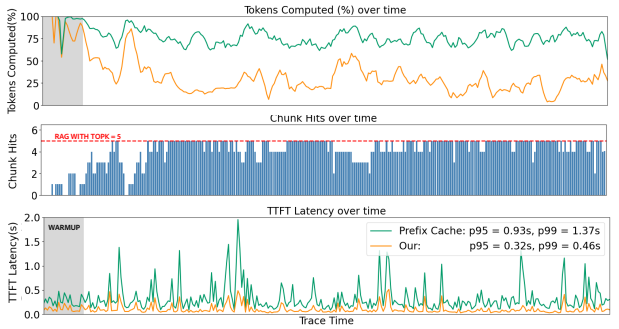


Figure 24: Evaluation of Cache-Craft on Sys-X trace on LLaMA-3-8B

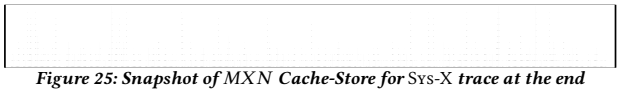


Figure 25: Snapshot of MXN Cache-Store for Sys-X trace at the end

for LLaMA-3-8B with batch-size= 4. We can see Cache-Craft outperforms all baselines across different sequence lengths and for 16k length it is 1.7× faster than Full-Recomp. In Fig. 23d we show for LLaMA-3-8B as we increase batch-size, Cache-Craft is more effective in controlling TTFT latency increases than all other methods because it requires much less recomputation to maintain quality.

6 Ablations and Discussions

Design Components in Cache-Craft: The ablation study in Fig. 26 on 2wikiMQA using LLaMA-3-8B highlights the impact of various design elements in Cache-Craft. We obtain a baseline score of 0.665 from Full KV Cache reuse with fixed RPE.

Removing components of our recomputation logic-specifically, β, Cache Context Index (*CCI*), and focus chunking—provides insights into performance dynamics. Removing β increases recomputation to 54% without improving quality, emphasizing its role in minimizing unnecessary recomputation for well-matched chunks. Disabling focus chunking similarly raises recomputation to 70% with minimal quality gains, underscoring the importance of both β and focus in optimizing recompute efficiency. Additionally, when fixed recomputation is applied without *CCI* (via random selection), quality declines dramatically to a ROUGE score of 0.73.

We also explore varying α values from 0.5 to 3, revealing an increasing quality trend: 0.825 for $\alpha = 0.5$, 0.896 for $\alpha = 1$, 0.94 for $\alpha = 2$, and 0.953 for $\alpha = 3$. However, this trend indicates diminishing returns as recomputation increases, highlighting a saturation point.

Context Size (Number of Chunks vs. Chunk Size): We analyze quality (ROUGE F1) trends with context lengths by varying chunk sizes (brown line) and the number of chunks (blue line) using LLaMA-80B with 30% recomputation, as shown in Fig. 27. The brown line demonstrates that quality consistently increases with larger chunk sizes, stabilizing around 0.92, which underscores the reliability of our recomputation logic for longer contexts. The blue line, representing quality with more chunks, exhibits a similar upward trend but slightly declines after saturation (approximately 0.91). This drop, highlighted in red on the plot, indicates that focus chunk selection becomes less effective with too many chunks. Notably, when the "*focused* chunks" filter is disabled (indigo line), quality remains stable, suggesting that the decline is attributed to the error from the "*focused* chunks" selection mechanism.

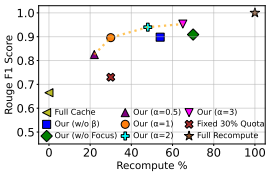


Figure 26: Design elements on Rouge F1 vs. recompute tokens for 2wikiMQA with LLaMA-3-8B and 30% recomp. The dotted line connects different α. α=1 w/o (β and focus)

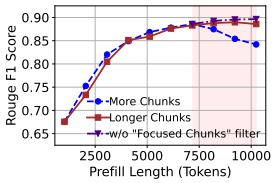


Figure 27: ROUGE-F1 quality for context length, measured by varying chunk sizes (brown) and number of chunks (blue), with LLaMA-3-8B at 30% recomputation.

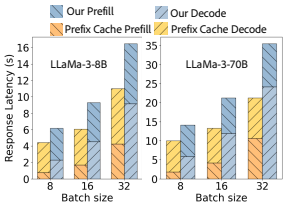


Figure 28: Prefill and decode latencies across batch size for LLaMA-3 8B and 70B/ Prefill takes up an increasing proportion of total time for larger batch sizes.

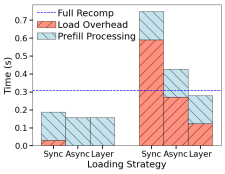


Figure 29: Cache loading overhead in CACHE-CRAFT from different memory hierarchies using LLaMA-3-8B with and w/o preloading.

RPE	Causal	Rouge
×	×	0.15
×	✓	0.22
✓	×	0.66
✓	✓	**0.89**

Table 3: Impact of fixing only RPE/-Causality and both RPE + Causality for chunk-cache reuse for 2WikiMQA

Why caching chunks is acceptable in practice? Our evaluations in Fig. 20 and the qualitative user study in Table 2 demonstrate that recomputing attention scores for only 30% of carefully selected tokens while using precomputed caches for the rest achieves 93% of the user score attained by full attention computation while significantly improving performance. This is driven by two observations:

(1) Retrieved RAG chunks, such as document sections in Sys-X, typically exhibit low inter-dependencies, as attention scores decline with token distance. For large chunks (>883 tokens) in Sys-X, the intra-attention is **2.18x** inter-attention on average with only **23%** of chunks being highly contextualized. To address such chunks, CACHE-CRAFT selectively recomputes KV caches for a few contextualized tokens, producing outputs close to ideal (§3.2). Fig. 20 shows CACHE-CRAFT offers the best trade-off between recomputation budget and quality, outperforming all baselines with a ROUGE F1 score of **0.87** using 20% recomputation for Sys-X while a threshold of **0.7** is considered good for semantic similarity [47]. This is also supported by our user study results which show **78.9%** user acceptance for CACHE-CRAFT versus **83.7%** for vanilla vLLM with exact computation using LLaMa-3-8B (Table 2).

(2) TTFT metric is critical in production. Current methods like prefix-caching suffer under heavy load, with TTFT reaching **35s** for LLaMa-3-70B on 4 A100 GPUs (Fig. 28). The proportion of TTFT in overall response latency increases with the higher system-load. CACHE-CRAFT reduces TTFT latency by independent caching of prior context. Unlike prefix caching, which has low hit rates and high memory overhead (Fig. 5a), CACHE-CRAFT stores chunks independently, allowing to store more chunks in HBM and achieving higher cache hit rates by reusing chunks in different combinations.

Approximation (Position vs. Causal): In Table 3, we observe that reusing caches from non-prefix chunks significantly degrades performance, resulting in a ROUGE F1 score of 0.15 when neither RPE nor causality is fixed. Fixing causality without adjusting RPE yields minimal improvement (0.22) while optimizing RPE alone achieves 0.665, which serves as the FULL-CACHE baseline. Notably, CACHE-CRAFT achieves a ROUGE score of 0.896 with just 30% recomputation, demonstrating that correct positional encoding combined with selective recomputation can effectively approximate the benefits of full recomputation, which scores 1.0.

7 Related Works

LLM Serving Efficiency: Multiple works looked into achieving service level objectives (SLOs) at scale[14, 15, 31, 67]. The works in [17, 42, 68] looked into optimizations of memory, while [34, 79, 81] have explored parallelism and batching. [19, 27, 41] aimed to improve the KV computations, primarily using the model's sparsity.

Context Compression and KV Cache Reduction: Improving decode speed of LLMs is a widely studied field. Several system-level techniques to optimize the prefill have been adopted [32, 33, 62]. These works primarily aim to reduce the size of the KV cache during generation. Works like [20, 38, 39] focused on compressing the context length to optimize the prefill. Some other similar works [2, 54, 82] drop unimportant tokens while a few modify attention or use gist tokens to achieve KV reduction [57, 78]. Another set of works that aim at KV reuse has enabled increased decoding speed by saving redundant computation. Most of these assume prefix-match [28, 40, 52, 53, 84], which is impractical for RAG-systems. Although *Prompt Cache*[30] enables KV cache reuse at various positions, it struggles to maintain satisfactory generation quality due to inaccuracies in positional encoding and a lack of consideration for cross-attention. CACHE-CRAFT enables efficient KV-cache reuse for RAG without compromising quality.

KV Cache Quantization and Management: Quantization of KV-cache reduces computation while maintaining generation quality [22, 36]. Some works address fitting large KV caches in limited memory [42, 43, 65, 74, 75]. vLLM [42] reduces KV cache due to fragmentation. *Prompt Cache* [30] reuses KV-caches at different positions but relies on a rigid prompt structure, leading to poor quality when the structure changes. These orthogonal techniques can complement CACHE-CRAFT for additional efficiency.

Approximation in Systems: Controlled approximation in KV-cache computation is inspired by prior works that used approximation techniques in image generation [4, 46, 55, 56], data analytics [3, 6, 29, 61], and video analytics [77, 80].

8 Conclusion

We introduced CACHE-CRAFT, a system that efficiently manages precomputed states corresponding to the chunks of the knowledge base for RAG. We presented several in-depth analyses of real production workloads showing several interesting characteristics of RAG-systems that show significant opportunities for *chunk-cache* reuse but also highlight the technical challenges. With our novel technique for identifying reusable chunks, selective recompute, and cache management policies, we show that CACHE-CRAFT can provide a significant speed-up without compromising the quality for real production workloads as well as popular RAG datasets.

References

[1] [n. d.]. Amazon EC2 P4d Instances – AWS. https://aws.amazon.com/ec2/instance-types/p4/. (Accessed on 10/18/2024).

[2] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: Kv cache reduction through key tokens selection for efficient generative inference. *Proceedings of Machine Learning and Systems* 6 (2024), 114–127.

[3] Shubham Agarwal, Gromit Yeuk-Yin Chan, Shaddy Garg, Tong Yu, and Subrata Mitra. 2023. Fast Natural Language Based Data Exploration with Samples. In *Companion of the 2023 International Conference on Management of Data*. 155–158.

[4] Shubham Agarwal, Subrata Mitra, Sarthak Chakraborty, Srikrishna Karanam, Koyel Mukherjee, and Shiv Kumar Saini. 2024. Approximate Caching for Efficiently Serving {Text-to-Image} Diffusion Models. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1173–1189.

[5] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} Tradeoff in {LLM} Inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[6] Ghazi Shazan Ahmad, Shubham Agarwal, Subrata Mitra, Ryan Rossi, Manav Doshi, Vibhor Porwal, and Syam Manoj Kumar Paila. 2024. ScaleViz: Scaling Visualization Recommendation Models on Large Data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 93–104.

[7] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. 2024. Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 318–334. https://doi.org/10.1145/3617232.3624849

[8] Samaneh Aminikhanghahi, Tinghui Wang, and Diane J Cook. 2018. Real-time change point detection with application to smart home time series data. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2018), 1010–1023.

[9] AI Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card* 1 (2024).

[10] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2023. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508* (2023).

[11] Anthony Chen, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Evaluating question answering evaluation. In *Proceedings of the 2nd workshop on machine reading for question answering*. 119–124.

[12] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

[13] Vincent A Cicirello. 2019. Kendall tau sequence distance: Extending Kendall tau from ranks to sequences. *arXiv preprint arXiv:1905.02752* (2019).

[14] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. 2020. InferLine: ML Prediction Pipeline Provisioning and Management for Tight Latency Objectives. arXiv:1812.01776 [cs.DC]

[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

[16] Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonellotto, and Fabrizio Silvestri. 2024. The power of noise: Redefining retrieval for rag systems. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 719–729.

[17] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[19] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems* 35 (2022), 30318–30332.

[20] Harry Dong, Xinyu Yang, Zhenyu Zhang, Zhangyang Wang, Yuejie Chi, and Beidi Chen. 2024. Get More with LESS: Synthesizing Recurrence with KV Cache Compression for Efficient LLM Inference. *arXiv preprint arXiv:2402.09398* (2024).

[21] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[22] Shichen Dong, Wen Cheng, Jiayu Qin, and Wei Wang. 2024. QAQ: Quality Adaptive Quantization for LLM KV Cache. *arXiv preprint arXiv:2403.04643* (2024).

[23] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161* (2019).

[24] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[25] Karima Echihabi. 2020. High-dimensional vector similarity search: from time series to deep network embeddings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2829–2832.

[26] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New trends in high-d vector similarity search: al-driven, progressive, and distributed. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3198–3201.

[27] Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*. PMLR, 10323–10337.

[28] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. AttentionStore: Cost-effective Attention Reuse across Multi-turn Conversations in Large Language Model Serving. *arXiv preprint arXiv:2403.19708* (2024).

[29] Minos N Garofalakis and Phillip B Gibbons. 2001. Approximate Query Processing: Taming the TeraBytes.. In *VLDB*, Vol. 10. 645927–672356.

[30] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.

[31] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[32] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[33] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. 2021. ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 692–705.

[34] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[35] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. 2020. Constructing A Multi-hop QA Dataset for Comprehensive Evaluation of Reasoning Steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, Donia Scott, Nuria Bel, and Chengqing Zong (Eds.). International Committee on Computational Linguistics, Barcelona, Spain (Online), 6609–6625. https://doi.org/10.18653/v1/2020.coling-main.580

[36] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079* (2024).

[37] GI Ivchenko and SA Honov. 1998. On the jaccard similarity test. *Journal of Mathematical Sciences* 88 (1998), 789–794.

[38] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. Llmlingua: Compressing prompts for accelerated inference of large language models. *arXiv preprint arXiv:2310.05736* (2023).

[39] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839* (2023).

[40] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *arXiv preprint arXiv:2404.12457* (2024).

[41] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. 2022. A fast post-training pruning framework for transformers. *Advances in Neural Information Processing Systems* 35 (2022), 24101–24116.

[42] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[43] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[44] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. 2022. xFormers: A modular and hackable Transformer modelling library. https://github.com/facebookresearch/xformers.

[45] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks.

Advances in Neural Information Processing Systems 33 (2020), 9459–9474.

[46] Muyang Li, Tianle Cai, Jiaxin Cao, Qinsheng Zhang, Han Cai, Junjie Bai, Yangqing Jia, Kai Li, and Song Han. 2024. Distrifusion: Distributed parallel inference for high-resolution diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7183–7193.

[47] Shuo Li, Sangdon Park, Insup Lee, and Osbert Bastani. 2024. TRAQ: Trustworthy Retrieval Augmented Question Answering via Conformal Prediction. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Kevin Duh, Helena Gomez, and Steven Bethard (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 3799–3821. https://doi.org/10.18653/v1/2024.naacl-long.210

[48] Tatiana Likhomanenko, Qiantong Xu, Gabriel Synnaeve, Ronan Collobert, and Alex Rogozhnikov. 2021. Cape: Encoding relative positions with continuous augmented positional embeddings. *Advances in Neural Information Processing Systems* 34 (2021), 16079–16092.

[49] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[50] Chin-Yew Lin and FJ Och. 2004. Looking for a few good metrics: ROUGE and its evaluation. In *Ntcir workshop*.

[51] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[52] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. 2024. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821* (2024).

[53] Yuhan Liu, Hanchen Li, Kuntai Du, Jiayi Yao, Yihua Cheng, Yuyang Huang, Shan Lu, Michael Maire, Henry Hoffmann, Ari Holtzman, et al. 2023. Cachegen: Fast context loading for language model applications. *arXiv preprint arXiv:2310.07240* (2023).

[54] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2024. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems* 36 (2024).

[55] Chen-Yi Lu, Shubham Agarwal, Md Mehrab Tanjim, Kanak Mahadik, Anup Rao, Subrata Mitra, Shiv Kumar Saini, Saurabh Bagchi, and Somali Chaterji. 2024. RECON: Training-Free Acceleration for Text-to-Image Synthesis with Retrieval of Concept Prompt Trajectories. In *European Conference on Computer Vision*. Springer, 288–306.

[56] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2024. Deepcache: Accelerating diffusion models for free. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 15762–15772.

[57] Jesse Mu, Xiang Li, and Noah Goodman. 2024. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems* 36 (2024).

[58] Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. 2016. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023* (2016).

[59] Shashi Narayan, Shay B Cohen, and Mirella Lapata. 2018. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745* (2018).

[60] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel. 2014. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 76–85.

[61] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*. 1461–1476.

[62] Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. 2022. Dota: detect and omit weak attentions for scalable transformer acceleration. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 14–26.

[63] P Rajpurkar. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[64] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[65] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[66] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2020. INFaaS: A Model-less and Managed Inference Serving System. arXiv:1905.13348 [cs.DC] https://arxiv.org/abs/1905.13348

[67] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[68] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. 2023. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 701–718.

[69] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[70] Philippe Tillet. 2021. Triton: Open-source GPU programming for neural networks. https://openai.com/index/triton/.

[71] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[72] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. 2022. ♪ MuSiQue: Multihop Questions via Single-hop Question Composition. *Transactions of the Association for Computational Linguistics* 10 (05 2022), 539–554. https://doi.org/10.1162/tacl_a_00475 arXiv:https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00475/2020694/tacl_a_00475.pdf

[73] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[74] Kefei Wang and Feng Chen. 2023. Catalyst: Optimizing Cache Management for Large In-memory Key-value Systems. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4339–4352.

[75] Kefei Wang, Jian Liu, and Feng Chen. 2020. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[76] Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023. Learning to filter context for retrieval-augmented generation. *arXiv preprint arXiv:2311.08377* (2023).

[77] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. 2018. {VideoChef}: Efficient Approximation for Streaming Video Processing Pipelines. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 43–56.

[78] Yu Yan, Jiusheng Chen, Weizhen Qi, Nikhil Bhendawade, Yeyun Gong, Nan Duan, and Ruofei Zhang. 2021. El-attention: Memory efficient lossless attention for generation. In *International Conference on Machine Learning*. PMLR, 11648–11658.

[79] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[80] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. 2017. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 377–392.

[81] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 787–808.

[82] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. 2024. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems* 36 (2024).

[83] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P Xing, et al. 2023. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998* (2023).

[84] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently Programming Large Language Models using SGLang. (2023).